The Use of Explicit Plans to Guide Inductive Proofs

The Use of Explicit Plans to Guide Inductive Proofs *

Alan Bundy

Department of Artificial Intelligence
University of Edinburgh
Edinburgh, EH1 1HN, Scotland

Abstract

We propose the use of explicit proof plans to guide the search for a proof in automatic theorem proving. By representing proof plans as the specifications of LCF-like tactics, [Gordon *et al* 79], and by recording these specifications in a sorted meta-logic, we are able to reason about the conjectures to be proved and the methods available to prove them. In this way we can build proof plans of wide generality, formally account for and predict their successes and failures, apply them flexibly, recover from their failures, and learn them from example proofs.

We illustrate this technique by building a proof plan based on a simple subset of the implicit proof plan embedded in the Boyer-Moore theorem prover, [Boyer & Moore 79].

Space restrictions have forced us to omit many of the details of our work. These are included in a longer version of this paper which is available from: The Documentation Secretary, Department of Artificial Intelligence, University of Edinburgh, Forrest Hill, Edinburgh EH1 2QL, Scotland.

Key words and phrases. Proof plans, inductive proofs, theorem proving, automatic programming, formal methods, planning.

1 Introduction

In this paper we propose a new technique for guiding an automatic theorem prover in its search for a proof, namely the use of *explicit proof plans*. This proposal was motivated by a current research project in the mathematical reasoning group at Edinburgh to develop automatic search control for the NuPRL program synthesis system, [Constable *et al* 86], and it was inspired by an earlier project of the group on the use of *meta-level inference* to guide an equation solving system, PRESS, [Bundy & Welham 81].

NuPRL can prove theorems by mathematical induction. In fact, inductive proofs are required for the synthesis of recursive programs: the type of induction used determining the type of recursion synthesised. In logic and functional programs, recursion is used in place of the imperative program constructs of iteration, *eg* while, until, do, etc. We are thus particularly interested in inductive proofs. The best work to date on the guidance of inductive proofs is that by Boyer and Moore, [Boyer & Moore 79]. Figure 1 contains a simple example of the kind of inductive proof found by their theorem prover. Hence, we have been adapting the techniques embedded in the Boyer-Moore theorem prover to the NuPRL environment, [Stevens 87].

In order to adapt the Boyer-Moore work we first need to understand why it works. Their program contains a large amount of heuristic information which is highly successful in guiding inductive proofs. However, the descriptions of these heuristics in [Boyer & Moore 79] are not always clear about why they are successful nor why they are applied in a particular order. Some heuristics are not appropriate in the NuPRL system, or require modification to make them appropriate. Thus it is necessary to rationally reconstruct the Boyer-Moore work in order to apply it to another system.

But we want to go further than this. We want to give a formal account of the Boyer-Moore heuristics, from which we can predict the circumstances in which they will succeed and fail, and with which we can explain their structure and order. We also want to apply the heuristics in a flexible

*I am grateful for many long conversations with other members of the mathematical reasoning group, from which many of the ideas in this paper emerged. In particular, I would like to thank Frank van Harmelen, Jane Hesketh and Andrew Stevens for feedback on this paper. The research reported in this paper was supported by SERC grant GR/D/44874 and Alvey/SERC grant GR/D/44270.

way and to learn new ones from example proofs. To achieve these goals we intend to represent the Boyer-Moore heuristics in an explicit proof plan based on the ideas of meta-level inference.

2 Explicit Proof Plans

We are confident that we can find such an explicit proof plan for a number of reasons. We believe that human mathematicians can draw on an armoury of such proof plans when trying to prove theorems. It is our intuition that we do this when proving theorems, and the same intuition is reported by other experienced mathematicians. One can identify such proof plans by collecting similar proofs into families having a similar structure, *eg* those proved by 'diagonalization' arguments. Many inductive proofs seem to have such a similar structure (see section 4 below). Within such families one can distinguish 'standard' from 'interesting' steps. The standard ones are those that are in line with the plan and the interesting ones are those that depart from it. The Boyer-Moore program proves a large number of theorems by induction using the same heuristics. These proofs all seem to belong to the same family.

The properties we desire of the proof plans that we seek are as follows:

- **Usefulness**: The plan should guide the search for a proof to a successful conclusion.
- **Generality**: The plan should succeed in a large number of cases.
- **Expectancy**: The use of the plan should carry some expectation of success, *ie* we ought to have some story to tell about why the plan often succeeds, and to be able to use this to predict when it will succeed and when it will fail.
- **Uncertainty**: On the other hand, success cannot be guaranteed. We will want to use plans in undecidable areas. If our ability to predict its success or failure was always perfect then the plan would constitute a decision procedure — which is not possible.
- **Patchability**: It should be possible to patch a failed plan by providing alternative steps.
- **Learnability**: It should be possible automatically to learn new proof plans.

The above properties argue for an *explicit* representation of proof plans with which one can reason. The reasoning would be used to account for the probable success of the plan under certain conditions (expectancy), and to replan dynamically when the plan fails (patchability). The explicit representation would enable plans to be learnt (learnability). The uncertainty property is discussed in section 7.

We have chosen to represent our plans in a sorted meta-logic. This gives an explicit representation to reason with, and also allows the plans to be very general (generality), in contrast to plans which are merely sequences of object-level rule applications.

We now turn to a detailed investigation of the Boyer-Moore heuristics in an attempt to extract from them the explicit meta-level proof plan that we require.

3 A Typical Inductive Proof

In order to investigate the Boyer-Moore heuristics, it will be instructive to study a typical proof of the kind that these heuristics can construct. Figure 1 is such a proof: the associativity of $+$ over the natural numbers. The first line is a statement of the theorem. Each subsequent line is obtained by rewriting a subexpression in the line above it. The subexpression to be rewritten is underlined and the subexpression which replaces it is overlined. The (recursive) definition of $+$ is given in the small box.

The proof is by backwards reasoning from the statement of the conjecture. The first step is to apply the standard arithmetic induction schema to the theorem: replacing x by 0 in the base case and by $s(x)$ in the induction conclusion of the step case. The equations constituting the recursive definition of $+$ are then applied: the base equation to the base case and the step equation to the step case. Two applications of the base equation rewrite the base case to an equation between two identical expressions, which reduces to *true*. Three applications of the step equation raise the occurrences of the successor function, s, from their innermost positions around the xs to being the outermost functions of the induction conclusion. The two arguments of the successor functions are identical to the two arguments of $=$ in the induction hypothesis. The induction hypothesis is then used to substitute one of these arguments for the other in the induction conclusion, and the induction hypothesis is dropped. The two arguments of the successor functions are now identical and reduce to *true*.

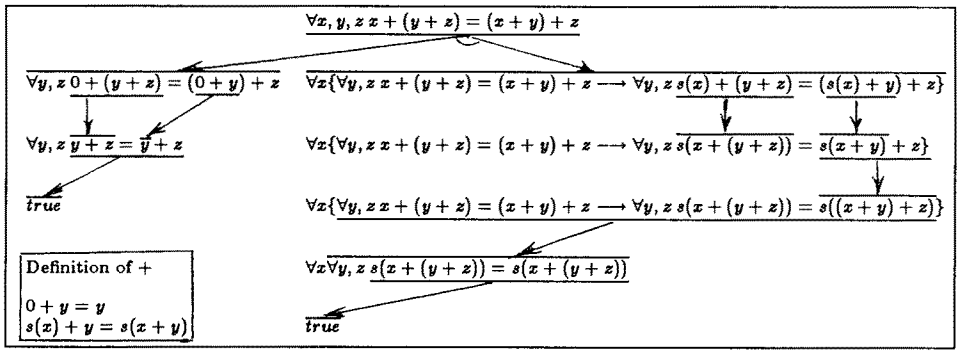

Figure 1: Proof of the Associativity of +

4 Simplified Boyer-Moore Proof Plan

We can pick out the general aspects of the proof in figure 1, and the above explanation of it, by displaying the schematic proof of figure 2. This schematic proof captures the spirit of the Boyer-Moore heuristics in a very simplistic way. Some of the extensions required to capture the full power of their theorem prover are discussed in the longer version of this paper.

In figure 2 capital letters indicate meta-variables. For instance, X and Y range over variables, F ranges over functions, and A, B_i, T_i, etc, range over terms. The difference between $T(X)$ and $T[X]$ is that the X in the round brackets signifies all occurrences of X in T whereas the X in the square brackets signifies some particular occurrence of X. Thus $T(Y)$ implies that all occurrences of X are replaced by Y, whereas $T[Y]$ implies that only one occurrence is replaced. In both cases the function or term may also contain variables other than X or Y. Note that the round bracket notation is unsound if the normal rules for substitution are applied to it. This is discussed further in the longer version of this paper.

Each arc is labelled with the name of the step that justifies the rewriting. Following LCF, [Gordon *et al* 79], we call these steps *tactics*. A $\surd$ sign beside a tactic indicates that it is guaranteed to succeed, whereas a ? indicates that it might fail. As in LCF, a tactic will be implemented as a program whose effect is to apply the appropriate rewritings to make the proof steps illustrated. However, whereas the primitive tactics provided in LCF apply only a small sequence of steps, we are also interested in designing tactics that will automatically complete a whole proof, or a substantial part of it, *ie* we are also interested in proof strategies.

Just as in the associativity proof of figure 1, the first tactic is to apply *induction*. Note that the induction scheme used, corresponds to the recursive scheme used to define F and that the induction variable to which it is applied is X, the variable in the recursive argument position of F. The major Boyer-Moore heuristic is to generalize this link between induction and recursion to most commonly occurring recursive data-structures and forms of recursion over them. The idea is to use the occurrence of recursive functions in the conjecture to suggest what induction scheme to use (one corresponding to the recursive structure of the function) and what variable(s) to induce on (those that occur in the recursive argument position(s) of the function). See [Stevens 87] for a more detailed analysis and rational reconstruction of this heuristic.

The equations that recursively define F are then applied to the base and step cases of the resulting formula, using the tactics *take-out* and *ripple-out*, respectively. The base case is simplified by this, but not solved as in the associativity proof. In the step case the occurrences of s are raised from their innermost positions to the outermost positions in the induction conclusion. The *ripple-out*[1] tactic does this using repeated applications of the step case of the recursive definition. This application of *ripple-out* is not guaranteed to succeed because the terms T_1, T_2 and B might not be of the right form. For a further description of this tactic and a definition of what form these terms must take for its success to be guaranteed, see figure 3. If *ripple-out* does succeed then *fertilization*[2] is

[1]The analogy is to a series of waves that carry the s from one place to another.

[2]The name is taken from Boyer and Moore. In the analogy the induction hypothesis is the sperm that fertilizes

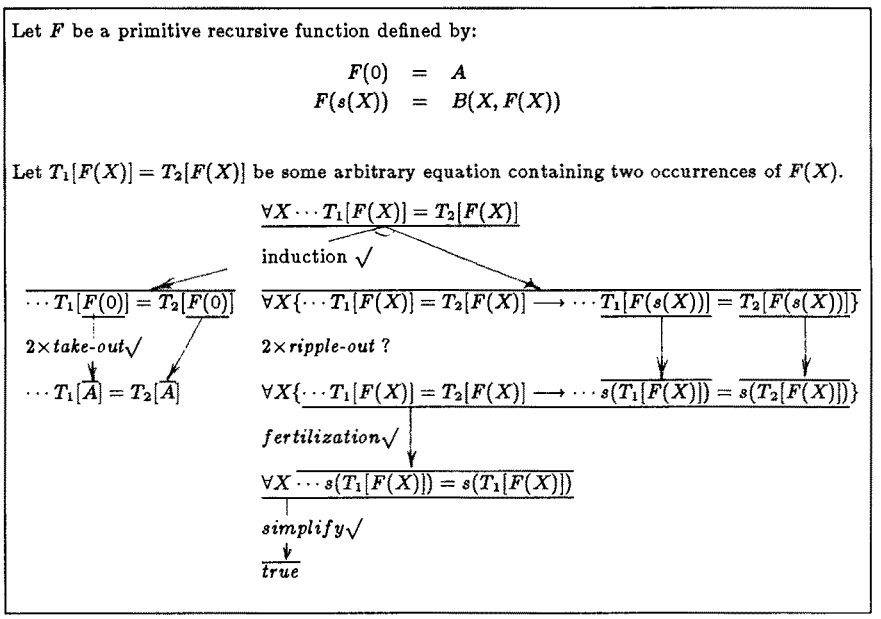

Figure 2: Simplified Boyer-Moore Proof Plan

guaranteed to succeed. It substitutes the T_1 term for the T_2 term in the induction conclusion and the step case reduces to *true* via an application of some simplifying rules like the reflexivity axiom.

Note that, unlike the associativity proof in figure 1, the final step of the general proof does not solve the problem. However, the general proof does exchange the original conjecture for a sub-goal from which all occurrences of the function F have been eliminated. This can be seen as the aim of the general proof. Repeated applications of it will cause recursively defined functions to be systematically eliminated from the current sub-goals and replaced by the functions by which they are defined. If defined functions are arranged in a hierarchy with each defined function ordered above those by which it is defined, and a highest function is eliminated on each round, then a set of sub-goals will eventually be generated in which only primitive (*ie* non-recursive) functions occur. The proof of these will not require induction.

In the sub-proof describing the *ripple-out* tactic given in figure 3, the terms T_1 and T_2 take the form of a nested chain of recursively defined functions, F_i, where each F_i appears in the recursive position of F_{i+1} and the definitions of the F_i are all very simple. The step equations merely ripple the occurrences of s out once. By applying these step equations repeatedly the occurrences of s are rippled out from their innermost to the outermost position. Following Darlington we call a single application of the step equation an $unfold$[3].

This version of *ripple-out* is very simple and special purpose. To make it more general, we need to extend it not only to constructor functions other than s, but also enable it to supplement the use of unfolding step equations with the application of lemmas of a similar syntactic form. This latter extension is discussed in section 7.

5 The Specification of Tactics with Methods

In LCF or NuPRL tactics can be implemented as ML programs which will guide the application of rewrite rules to control the search for a proof. We have begun just such an implementation of

the step conclusion. by making it provable.

[3]The analogy is with unfolding a piece of paper and taking out the present at the end.

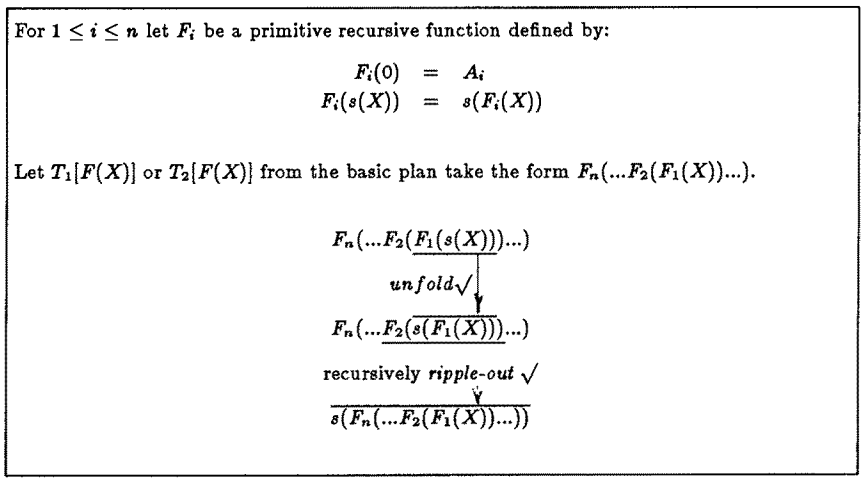

Figure 3: The Ripple-Out Sub-Plan

the Boyer-Moore heuristics, [Stevens 87]. However, such an implementation would not meet all the required properties of a proof plan. In particular, we require the ability to reason about the tactics in order to construct a proof plan for a problem and to replan when an existing proof plan fails. In order to conduct this reason we need to represent the conditions under which a tactic is applicable and the effect that it has if it succeeds, *ie* we need a *specification* of the tactic. Below we propose such a specification, which we plan to implement within the NuPRL or a similar framework.

Our specification formalism was adapted from that used in the LP system, [Silver 84], which was an extension of PRESS. The LP formalism was itself based on that of STRIPS. However, note that, unlike the plans formed in STRIPS-type plan formation, our plans will contain subroutines and recursion.

Following PRESS, we call the specification of a tactic, a *method*. A method is a frame containing information about the preconditions and effects of a tactic. A list of the slots in the frame and a description of the contents of each of them is given in table 1. Figure 4 is an example method for the *ripple-out* tactic. Methods for the other tactics in the simplified Boyer-Moore proof plan are given in the long version of this paper. Each of the slots contains a formula of our sorted meta-logic describing syntactic properties of the goal formulae before and after the tactic is applied. The meta-logical terms used in this paper are defined in table 2 and the sorts are defined in table 3. Definitions of the terms and sorts used in the remaining methods are given in the long version of this paper.

The description of the precondition is split between the input slot and the preconditions slot. The input slot contains a pattern which must match the before formula and the precondition contains additional information about the before formula which cannot be captured in this pattern. Similarly, the description of the effect is split between the output and the effects slots. The tradeoffs between representing information schematically in the input and output slots and representing it linguistically in the preconditions and effects slots, is discussed in the longer version of this paper.

A method represents an assertion in the meta-logic, namely that if a goal formula matches the input pattern and if the preconditions are true of it then the tactic is applicable. Furthermore, if the tactic application is successful then the resulting formula will match the output pattern and the effects will be true of it. The additional condition that the tactic application be successful means that the method is only a partial specification of the tactic. This is the key to the realisation of the uncertainty property and is discussed further in section 7 below.

The *ripple-out* tactic uses the repeated application of the unfold tactic to move the successor function from an innermost to an outermost position. *ripple-out* is illustrated in figures 1 and 3. The specification of the simple *ripple-out* tactic is given in table 4. The preconditions slot specifies that the input must be a nested sequence of simple recursive functions whose innermost argument

- **Name** - the name of the method. (We have followed the convention of using the tactic name for the method, augmented with additional arguments where necessary.)
- **Declarations** - a list of quantifier and sort declarations for meta-variables global to all the slots except the Tactics slot.
- **Input** - a schematic representation of the goal formula before the tactic applies.
- **Output** - a schematic representation of the goal formula after the tactic applies.
- **Preconditions** - a linguistic representation of further conditions required for the tactic to be applicable.
- **Effects** - a linguistic representation of additional effects of the method, including properties of the output and relationships between the input and output. They hold if the tactic applies.
- **Tactic** - a program for applying object-level rules of inference. This program is written in a subset of the same sorted meta-logic as the other slot values. This subset consists of applications of the object-level rules of inference and calls to sub-tactics. The tactic program serves also to specify the sub-tactics of this tactic and hence the sub-methods of this method. Meta-variables in this slot are local to each formula that constitutes the program[4].

Table 1: The Slots of a Method

has s as its dominant function. The output slot gives a pattern asserting that the output will an s whose argument is the input expression with the innermost s removed. No further effects information is required in this case. The tactic slot contains a recursively defined program, *ripple-out*, which takes a position and a formula and repeatedly applies $unfold$ from that position to the outermost position. A specification of an extended version of *ripple-out* is given in section 7.

6 The Use of Proof Plans

In this formalism a proof plan is the method for one of the top-level tactics, *ie* it is the specification of a strategy for controlling a whole proof, or a large part of one. This super-method is so constructed that the preconditions of each of its sub-methods are either implied by its preconditions or by the effects of earlier sub-methods. Similarly, its effects are implied by the effects of its sub-methods. If the preconditions of a method are satisfied then its tactic is applicable. If the tactic application succeeds[5] then its effects are satisfied. The original conjecture should satisfy the preconditions of the plan; the effects of the plan should imply that the conjecture has been proved.

We can formalize this argument by associating with each method a formula of the form:

$$\forall O \in OSort.\ declarations(M)\{preconditions(M, input(M)) \wedge name(M, input(M)) = O \longrightarrow effects(M, input(M), O) \wedge output(M) \equiv O\}$$

where M is a method name, $Slotname(M, ...)$ means the contents of slot $Slotname$ of method M applied to additional arguments ..., and $\equiv$ means syntactic identity. This formula can be read as asserting that if the input of a method satisfies the preconditions and if the tactic succeeds when applied to this input then the tactic's output matches the output slot and satisfies the effects of the method. We will call it the *expectancy* formula of the method, because it formalises our expectation that the method will do what it is intended to do.

Given, as an axiom, the expectancy formula for each of the sub-methods of a plan, axioms consisting of each of the tactic definitions, and various other axioms defining the meta-level terms of table 2, we can then prove as a theorem the expectancy formula for the super-method. I have carried out this programme for earlier versions of the methods described in section 5, *ie* given the expectancy formula for the method $unfold$ I have proved the expectancy formula for the method *ripple-out*, then given them for: *induction*, *take-out*, $fertilization$ and $simplify$, I have proved one for *basic-plan* (see [Bundy 87] for details). The structure of each proof is:

[5]The failure of tactics is discussed in section 7.

- *exp-at*$(Exp, Posn)$ is the sub-expression in expression Exp at position $Posn$. Positions are list of numbers which define an occurrence of one expression within another. For instance, [2,1] is the position of the 2nd argument of the 1st argument, *eg* the x in $f(g(2,x),3)$. Note that the order of the list of numbers is the reverse of the usual convention. This simplifies some of the formulae in the sequel. We adopt the convention that 0 denotes the function symbol itself, so that [0,1] is the position of g in $f(g(2,x),3)$.
- *single-occ*$(SubExp, Posn, SupExp)$ means that $SupExp$ contains precisely one occurrence of $SubExp$ and that this is at position $Posn$.
- *replace*$(Posn, NewExp, SupExp)$ is the expression obtained from $SupExp$ by replacing the sub-expression at position, $Posn$ with $NewExp$.
- *simp-rec*(F, N) means F is a primitive recursive function whose Nth argument is the recursion argument, and whose step equation is of the simple form $F(s(X)) = s(F(X))$, where X is the Nth argument of $F(X)$
- *app*(L_1, L_2) is the result of appending list L_1 to list L_2.
- $[Hd|Tl]$ is the list obtained from putting a new element Hd on the front on the list Tl.

Table 2: The Terms Used in the Meta-Logic

- *exprs* is the set of all expressions.
- *terms* is the set of all terms.
- *nums* is the set of all natural numbers
- *posns* is the set of all lists of natural numbers.

Table 3: The Sorts Used in the Meta-Logic

super-method preconditions ⟶ sub-method preconditions
⟶ sub-method effects
⟶ super-method effects

The steps going between super- and sub-methods require the tactic definition for the appropriate link. The other step is by assumption. Such theorems prove that if the conjecture satisfies the preconditions of a plan and each of the sub-tactics succeed then the resulting formula will satisfy the effects of the plan.

The methods of section 5 were hand-coded to represent a rational reconstruction of a simple version of the implicit proof in the Boyer-Moore theorem prover. We are also interested in the use of the techniques of plan formation and/or automatic program synthesis to construct such proof plans automatically. Of relevance here is the work of Silver, [Silver 84], who developed the technique of *Precondition Analysis* for learning proof plans from examples in the domain of equation solving. Our representation of method is based on that of Precondition Analysis. Desimone has been extending this technique by removing some technical limitations which made it inapplicable to general proofs, [Desimone 87]. Precondition Analysis is also capable of learning new methods, *ie* the specifications of unknown tactics.

Also relevant is the work of Knoblock and Constable, [Knoblock & Constable 86], who have shown how NuPRL can be applied to the synthesis of its own tactics. We aim to explore this self-application of NuPRL to the generation of new tactics from the methods which specify them: methods which may have been learnt from example proofs using Precondition Analysis. Our representation of methods seems to be compatible with the specifications used by Knoblock and Constable.

7 How a Tactic may Fail

So far all the tactics that we have specified have been guaranteed to succeed provided their preconditions are met. Thus plans formed from them are guaranteed to succeed. However, as discussed in our

Name	*ripple-out*$(Posn)$
Declarations	$\forall Posn \in posns,\ \forall SExp \in exprs,\ \forall Exp \in exprs.$
Input	$SExp$
Output	$s(Exp)$
Preconditions	$SExp = replace(Posn, s(exp\text{-}at(Exp, Posn)), Exp)) \wedge$
	$\{\forall Front \in posns, \forall Back \in posns, \forall N \in nums$
	$app(Front, [N\|Back]) = Posn \longrightarrow simp\text{-}rec(exp\text{-}at(Exp, [0\|Back]), N)\}$
Effects	nil
Tactic	$ripple\text{-}out([], Expr) = Expr$
	$ripple\text{-}out([Hd\|Tl], Expr) = ripple\text{-}out(Tl, unfold([Hd\|Tl], Expr))$

Table 4: The Ripple Out Method

list of desired properties of a proof plan, we cannot in general expect proof plans to be guaranteed successful, particularly in undecidable areas. Thus we must expect tactics to fail sometimes.

An example of a tactic that can sometimes fail is the *ext-ripple-out* tactic specified by the method in table 5. This method is similar to the one for the *ripple-out* tactic except that:

- we no longer require a nested sequence of simple recursive functions in the input, but only a single primitive recursive one;
- the output is no longer of the very simple form, $s(Exp)$, but is some expression containing a single occurrence of Exp;
- the tactic is defined using an extended version of $unfold$, called *wavelet*;
- and there is an extra equation to deal with the case that the rippling out process peters out benignly.

The idea of *wavelet*, for which we have not given a method, is that it can apply not just the step equations of recursive definitions, but any rewrite rule of the syntactic form:

$$G(B_1(X)) \Rightarrow B_2(X, G(X)) \tag{1}$$

eg[6]

$$even(X + Y) \Rightarrow even(X) \wedge even(Y) \tag{2}$$

where G is *even*, $B_1(X)$ is $X + Y$, $B_2(X, Z)$ is $Z + even(Y)$[7] and *even* is defined by:

$$\begin{array}{rcl} even(0) & & \\ \neg even(s(0)) & & \\ even(s(s(x))) & \longleftrightarrow & even(x) \end{array}$$

Note that if $B_2(X, Z) \equiv Z$ then the rippling out will peter out benignly. The step equation for *even* is an example of such a rule.

$$even(s(s(X))) \Rightarrow even(X)$$

ext-ripple-out is able to ripple out expressions that the simple version cannot cope with. For instance, consider the expression:

$$even(s(x) \times y)$$

where $\times$ is defined by:

$$\begin{array}{rcl} 0 \times y & = & 0 \\ s(x) \times y & = & x \times y + y \end{array}$$

[6]Recall that we are reasoning backwards. The logical implication runs in the reverse direction.

[7]Recall also that our notation allows the term meta-variables, *eg* B_1, to contain variables, *eg* Y, other than those explicitly mentioned.

Name	$\mathit{ext\text{-}ripple\text{-}out}(\mathit{app}(\mathit{Posn}_2, \mathit{Posn}_1), \mathit{Exp})$
Declarations	$\forall \mathit{Posn}_1 \in \mathit{posns}$, $\forall \mathit{Posn}_2 \in \mathit{posns}$, $\forall \mathit{Exp} \in \mathit{exprs}$, $\forall \mathit{SExp} \in \mathit{exprs}$, $\forall \mathit{SExp}_2 \in \mathit{exprs}$.
Input	SExp
Output	$\mathit{SExp2}$
Preconditions	$\exists B \in \mathit{terms}\ \mathit{SExp} =$ $\mathit{replace}(\mathit{Posn}_1, \mathit{replace}(\mathit{Posn}_2, \mathit{exp\text{-}at}(\mathit{Exp}, \mathit{Posn}_1), B), \mathit{Exp}))$
Effects	$\exists \mathit{Posn}_3\ \mathit{single\text{-}occ}(\mathit{Exp}, \mathit{Posn}_3, \mathit{SExp2})$
Tactic	$\mathit{ext\text{-}ripple\text{-}out}([], \mathit{Expr}, \mathit{SExpr}) = \mathit{SExpr}$ $\mathit{ext\text{-}ripple\text{-}out}(\mathit{Psn}, \mathit{Expr}, \mathit{Expr}) = \mathit{Expr}$ $\mathit{wavelet}(\mathit{InPsn}, \mathit{SExpr}, \mathit{OutPsn}, \mathit{OutExpr})$ $\longrightarrow$ $\mathit{ext\text{-}ripple\text{-}out}(\mathit{InPsn}, \mathit{Expr}, \mathit{SExpr}) = \mathit{ext\text{-}ripple\text{-}out}(\mathit{OutPsn}, \mathit{Expr}, \mathit{OutExpr})$

Table 5: The Extended Ripple-Out Method

After one application of *wavelet* using the step equation of $\times$ we get the expression:

$$\mathit{even}(x \times y + y)$$

We can now use *wavelet* to apply rule 2 to get:

$$\mathit{even}(x \times y) \wedge \mathit{even}(y)$$

which contains $\mathit{even}(x \times y)$ as required.

However, the following expression also fits the preconditions of the tactic:

$$\mathit{even}(s(x))$$

but there is no rewrite rule of form 1 rule that matches this expression, so *wavelet* will fail, causing the failure of *ext-ripple-out*. Thus *ext-ripple-out* might fail even though its preconditions are satisfied because an appropriate rewrite rule is missing. This is a typical way in which tactics fail. This meets the uncertainty property of proof plans: a proof plan might fail even though its preconditions are satisfied because one of its sub-tactics fails.

One could argue that the preconditions of methods should be strengthened so that they implied the success of the tactic. However, note that, in practice, this would amount to running the tactic 'unofficially' in the precondition, to see if it succeeded, before running it 'officially'.

8 The Required Properties are Satisfied

In this section we return to the desired properties of proof plans given in section 2 and see that each of them has been met by our proposals.

- **Usefulness**: As the tactics run they will each perform a part of the object-level proof, so the plan guides the proof search.
- **Generality**: The proof plan formalism is not restricted to describing a sequence of object-level rule applications. Meta-level specifications can describe a large set of rules. The powerful tactic language can combine these by sub-routining, recursion, conditionals, etc.
- **Expectancy**: If the conjecture meets the preconditions of the plan and each tactic succeeds then the effects of the plan will be true and the conjecture will be proved. Thus the plan is expected to succeed.
- **Uncertainty**: However, a tactic may fail, causing failure of the plan, so the plan is not guaranteed to succeed.
- **Patchability**: Since the preconditions and effects of a failing tactic are known, plan formation and/or program synthesis techniques may be (re)used to patch the gap in the plan with a subplan. This could be done automatically and dynamically enabling the theorem prover to recover from a failed plan without having to throw away those parts of the current proof that did succeed.
- **Learnability**: An extended version of Silver's Precondition Analysis might be used to learn proof plans from example proofs. New methods might also be learnt by this technique and the tactics corresponding to these synthesised by a self-referential use of NuPRL.

9 Conclusion

In this paper we have explored the use of proof plans to guide the search for a proof in automatic theorem proving. We have advocated the explicit representation of proof plans in a sorted meta-logic. We have developed a formalism for representing such proof plans as the specifications of LCF-like tactics. This proposal has been illustrated by developing a proof plan for inductive proofs based on the work of Boyer and Moore and others. We have developed tactics for running this proof plan and methods which specify each of them.

The domain of inductive proofs has proved a productive one since there is a rich store of heuristic knowledge available on how to guide such proofs. We have used this heuristic knowledge in the design of our tactics and methods. Our formalism has enabled us to explain why this heuristic knowledge is successful (when it is) and why it fails (when it does). In fact, we can give formal proofs that certain preconditions are sufficient for success, albeit in a very simple case. We have thus provided an analysis of the Boyer-Moore theorem prover which is serving as a good basis for extending and improving their ideas and for transporting them to a different system (NuPRL).

Our explicit representation suggests techniques for the dynamic construction of proof plans. We hope it will be possible to use these to recover from failure by constructing an alternative sub-plan to fill the gap left by a failed tactic. We are also exploring the use of these techniques to learn new proof plans from examples of successful proofs.

A major goal is the extension of the simple plans described above to incorporate some of the extensions described in the long version of this paper. This involves the identification of new meta-level relations, properties and functions, *eg* to describe a *wavelet* rule, and their incorporation in the meta-logic. We also intend to implement the ideas described in this paper by extending our version of the NuPRL system to use these proof plans for guiding search, recovering from failure and learning from examples.

References

[Boyer & Moore 79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[Bundy & Welham 81] A. Bundy and B. Welham. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence*, 16(2):189–212, 1981. Also available as DAI Research Paper 121.

[Bundy 87] A. Bundy. *The derivation of tactic specifications.* Blue Book Note 356, Department of Artificial Intelligence, March 1987.

[Constable *et al* 86] R.L. Constable, S.F. Allen, H.M. Bromley, *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Desimone 87] R.V. Desimone. Learning control knowledge within an explanation-based learning framework. In I. Bratko and N. Lavrač, editors, *Progress in Machine Learning – Proceedings of 2nd European Working Session on Learning, EWSL-87, Bled, Yugoslavia*, Sigma Press, May 1987. Also available as DAI Research Paper 321.

[Gordon *et al* 79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[Knoblock & Constable 86] T. B. Knoblock and R.L. Constable. Formalized metareasoning in type theory. In *Proceedings of LICS*, pages 237–248, IEEE, 1986.

[Silver 84] B. Silver. Precondition analysis: learning control information. In *Machine Learning 2*, Tioga Publishing Company, 1984.

[Stevens 87] A. Stevens. *A Rational Reconstruction of Boyer-Moore Recursion Analysis.* Research Paper forthcoming, Dept. of Artificial Intelligence, Edinburgh, 1987. submitted to ECAI-88.